Invariant Generation in Vampire*

Kryštof Hoder[1], Laura Kovács[2], and Andrei Voronkov[1]

[1] University of Manchester
[2] TU Vienna

Abstract. This paper describes a loop invariant generator implemented in the theorem prover Vampire. It is based on the symbol elimination method proposed by two authors of this paper. The generator accepts a program written in a subset of C, finds loops in it, analyses the loops, generates and outputs invariants. It also uses a special consequence removal mode added to Vampire to remove invariants implied by other invariants. The generator is implemented as a standalone tool, thus no knowledge of theorem proving is required from its users.

1 Introduction

In [9] a new *symbol elimination* method of loop invariant generation was introduced. The method is based on the following ideas. Suppose we have a loop L with a set of (scalar and array) variables V. The set V defines the *language* of L. We extend the language L to a richer language L' by a number of functions and predicates. For every scalar variable v of the loop we add to L' a unary function $v(i)$ which denotes the value of v after i iterations of L, and similar for array variables. Thus, all loop variables are considered as functions of the loop counter. Further, we add to L' so-called *update predicates* expressing updates to arrays as formulas depending on the loop counter. After that we automatically generate a set P of first-order properties of the loop in the language L'. These properties are valid properties of the loop, yet they are not loop invariants since they use the extended language L'.

Note that any logical consequence of P that only contains variables in L is also a loop invariant. Thus, we are interested in finding logical consequences of formulas in P expressed in L. To this end, we run a first-order theorem prover using a saturation algorithm on P in such a way that it tries to derive formulas in L. To obtain a saturation algorithm specialised to efficiently derive consequences in L, we enhanced the theorem prover Vampire [7] by so-called *colored proofs* and a *symbol elimination mode*. In colored proofs, some (predicate and/or function) symbols are declared to have colors, and every proof inference can use symbols of at most one color.

As reported in [9], we tested Vampire on several benchmarks for invariant generation. It was shown that symbol elimination can infer complex properties with quantifier alternations. Symbol elimination thus provides new perspectives in automating program verification, since such invariants could not be automatically derived by other methods.

As the method is new, its practical power and limitations are not well-understood. The main obstacle to its experimental evaluation lies in the fact that program analysis and generation of input for symbol elimination by a separate tool is error-prone and

* Kryštof Hoder is supported by the School of Computer Science at the University of Manchester. Laura Kovács is supported by an FWF Hertha Firnberg Research grant (T425-N23). Andrei Voronkov is partially supported by an EPSRC grant. This research was partly supported by Dassault Aviation.

requires full knowledge of our invariant generation method. The tool described in this paper was designed with the purpose of creating a *standalone tool implementing invariant generation* by symbol elimination. Vampire can still be used for symbol elimination only, so that the program analysis is done by another tool (for example, for experiments with variations of the method).

The purpose of this paper is to describe the program analyser and loop invariant generator of Vampire, their implementation and use. We do not overview Vampire itself.
Related work. Reasoning about loop invariants is a challenging and widely studied research topic. We overview only some papers most closely related to our tool.

Automatic loop invariant generation is described in a number of papers, including [3, 12, 5, 10, 4, 6]. In [12] loop invariants are inferred by predicate abstraction over a set of a priori defined predicates, while [4] employs constraint solving over invariant templates. Input predicates in conjunction with interpolation are used to infer invariants in [10]. Unlike these works, we require no used guidance in providing input templates and/or predicates. User guidance is also not required in [3, 5], but invariants are derived using abstract interpretation [3, 5] or symbolic computation [6]. However, these approaches can only infer universally quantified invariants, whereas we can also derive invariants with quantifier alternation.

Our work is also related to first-order theorem proving [11, 13, 8]. These works implement superposition calculi, with a limited support for theories. However, only Vampire implements colored proofs and consequence removal essential for the symbol elimination method.

A more complex and general framework for program analysis is given in, e.g., [1, 2]. Whereas in [1, 2] theorem proving is integrated in a program analysis environment, we integrate program analysis in a theorem proving framework. Although our approach at the moment is limited to the analysis of a restricted class of loops, we are able to infer richer and more complex quantified invariants than [1, 2]. Combining our method with other techniques for verification and invariant generation is left for further work.

2 Invariant Generation in Vampire: Overview

To create an integrated environment for invariant generation, we implemented a simple program analyser and several new features in Vampire. The workflow of the invariant generation process is given in Figure 1.

The analyser itself comprises about 4,000 lines of C++ code (all Vampire is written in C++). In addition to the analyser, we had to extend formulas and terms with if-then-else and let-in constructs, implement colored proofs, automatic theory loading, and the consequence removal mode. All together making Vampire into an invariant generator required about 12,000 lines of code. Currently, the analyser only generates loop properties for symbol elimination, but we plan to use it in the future for a more powerful integration of program analysis and theorem proving.
Program analysis. The program analysis part works as follows. First, it extracts all loops from the input program. It ignores nested loops and performs the following steps on every non-nested loop.

1. Find all loop variables.
2. Classify them into variables updated by the loop and constant variables.

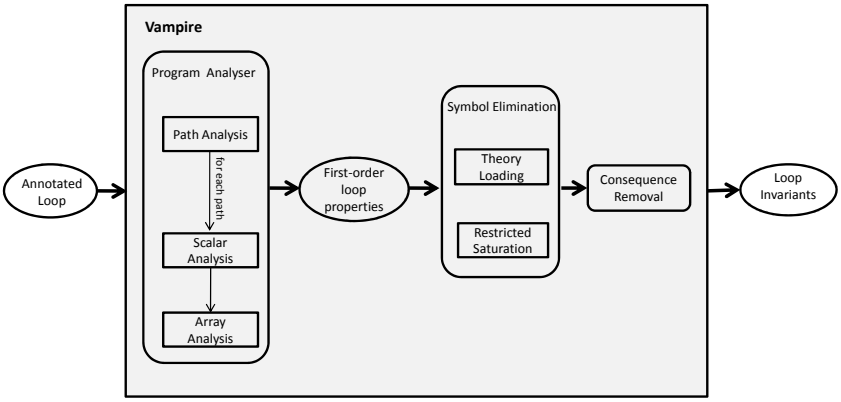

Fig. 1. Program Analysis and Invariant Generation in Vampire

3. Find counters, that is, updated scalar variables that are only incremented or decremented by constant values. Note that expressions used as array indexes in loops are typically counters.
4. Save properties of counters.
5. Generate update predicates of updated array variables and save their properties.
6. Save the formulas corresponding to the transition relation of the loop.
7. Generate a symbol elimination task for Vampire.

The input to the analyser is a program written in a subset of C. The subset consists of scalar integer variables, array variables, arithmetical expressions, assignments, conditionals and while-do loops. Nested loops are not yet handled.

Symbol Elimination and Theory Loading. The program analyser generates a set of first-order loop properties and information about which symbols should be eliminated. A (predicate and/or function) symbol is to be eliminated in Vampire whenever it is specified to have some color. The next phase of invariant generation runs symbol elimination on the set of formulas generated by the analyser. Before doing symbol elimination, Vampire checks which theory symbols (such as integer addition) are used and loads axioms relevant to these theory symbols. Theory symbols have no color in Vampire. After theory loading, Vampire runs a saturation algorithm on the theory axioms and the formulas generated by its analyser. A special term ordering is used to ensure that symbol elimination is effective and efficient.

Consequence Removal. The result of the symbol elimination phase is a set of loop invariants. This set is sometimes too large. For example, it is not unusual that Vampire generates over a hundred invariants in less than a second.

An analysis of these invariants shows that some invariants are concise and natural for humans, while some other invariants look artificial (this does not mean they are not interesting and/or not useful). It is typically the case that the generated set of invariants contains many invariants implied by other invariants in the set.

The next phase of invariant generation prunes the generated set by removing the implied invariants. Checking whether each generated invariant is implied by all other invariants is too inefficient. To remove them efficiently, we implemented a special consequence removal mode. The output of the tool is the set of all non-removed invariants.

Implementation and Availability. We implemented our approach to invariant generation in Vampire. The new version of Vampire is available from http://www.vprover.org. The current version of Vampire runs under Windows, Linux and MacOS.

Experiments. We evaluated invariant generation in Vampire using two benchmark suites: (1) challenging loops taken from [3, 12], and (2) a collection of 38 loops taken from programs provided by Dassault Aviation. We used a computer with a 2GHz processor and 2GB RAM and ran experiments using Vampire version 0.6. The symbol elimination phase was run with a 1 second time limit and the consequence removal phase with a 20 seconds time limit.

For all the examples the program analyser took essentially no time. It turned out that symbol elimination in one second can produce a large amount of invariants, ranging from one to hundreds. Consequence removal normally deletes about 80% of all invariants.

3 Conclusion

It is not unusual that program analysers call theorem provers or contain theorem provers as essential parts. Having a program analyser as part of a theorem prover is less common. We implemented an extension of Vampire by program analysis tools, which resulted in a standalone automatic loop invariant generator. Our tool derives logically complex invariants, strengthening the state-of-the-art in reasoning about loops.

References

1. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec♯ Programming System: An Overview. In *Proc. of CASSIS*, volume 3362 of *LNCS*, 2004.
2. L. Correnson, P. Cuoq, A. Puccetti, and J. Signoles. *Frama-C User Manual*. CEA LIST, 2010.
3. D. Gopan, T. W. Reps, and S. Sagiv. A Framework for Numeric Analysis of Array Operations. In *Proc. of POPL*, pages 338–350, 2005.
4. A. Gupta and A. Rybalchenko. InvGen: An Efficient Invariant Generator. In *Proc. of CAV*, pages 634–640, 2009.
5. N. Halbwachs and M. Peron. Discovering Properties about Arrays in Simple Programs. In *Proc. of PLDI*, pages 339–348, 2008.
6. T. Henzinger, T. Hottelier, L. Kovacs, and A. Rybalchenko. Aligators for Arrays. In *Proc. of LPAR-17*, volume 6355 of *LNAI*, pages 103–118, 2010.
7. K. Hoder, L. Kovacs, and A. Voronkov. Interpolation and Symbol Elimination in Vampire. In *Proc. of IJCAR*, pages 188–195, 2010.
8. K. Korovin. iProver - An Instantiation-based Theorem Prover for First-order Logic (System Description). In *Proc. of IJCAR*, volume 5195 of *LNAI*, pages 292–298, 2009.
9. L. Kovacs and A. Voronkov. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In *Proc. of FASE*, pages 470–485, 2009.
10. K. L. McMillan. Quantified Invariant Generation Using an Interpolating Saturation Prover. In *Proc. of TACAS*, volume 4963 of *LNCS*, pages 413–427, 2008.
11. S. Schulz. System Description: E 0.81. In *Proc. of IJCAR*, volume 3097 of *LNAI*, pages 223–228, 2004.
12. S. Srivastava and S. Gulwani. Program Verification using Templates over Predicate Abstraction. In *Proc. of PLDI*, pages 223–234, 2009.
13. C. Weidenbach, R. A. Schmidt, T. Hillenbrand, R. Rusev, and D. Topic. System Description: SpassVersion 3.0. In *Proc. of CADE*, volume 4603 of *LNAI*, pages 514–520, 2007.